Introduction to coding

HTML Version: https://stirlingcodingclub.github.io/getting_started/notes.html

Brad Duthie

19 OCT 2022

The purpose of this introduction

The purpose here is to get readers past the initial learning curve of coding as quickly as possible. If you want to start coding for yourself, particularly in R for data analysis, but are not sure how, then read on. By the end of these notes, you should be able to navigate through the basic graphical user interface of Rstudio and write some basic lines of code. The goal is not to develop proficiency in coding or R yet, but to help you get to the point at which it is possible to write and run code, make coding mistakes, and learn from other researcher's code.

Why use the R programming language?

The computer programming language R is a powerful and very widely-used tool among scientists for analysing data. You can use it to analyse and plot data, run computer simulations, or even write slides, papers, or books. The R programming language is completely free and open source, as is the popular Rstudio software for using it. It specialises in statistical computing, which is part of the reason for its popularity among scientists.

Another reason for its popularity is its versatility, and the ease with which new techniques can be shared. Imagine that you develop a new method for analysing data. If you want other researchers to be able to use your method in their research, then you could write your own software from scratch for them to install and use. But doing this would be very time consuming, and a lot of that time would likely be spent writing the graphical user interface and making sure that your program worked across platforms (e.g., on Windows and Mac). Worse, once written, there would be no easy way to make your program work with other statistical software should you need to integrate different analyses or visualisation tools (e.g., plotting data). To avoid all of this, you could instead just present your new method for data analysis and let other researchers write their own code for implementing it. But not all researchers will have the time or expertise to do this.

Instead, R allows researchers to write new tools for data analysis using simple coding scripts. These scripts are organised into R packages, which can be uploaded by authors to the Comprehensive R Archive Network (CRAN), then downloaded by users with a single command in R. This way, there is no need for completely different software to be used for different analyses – all analyses can be written and run in R.

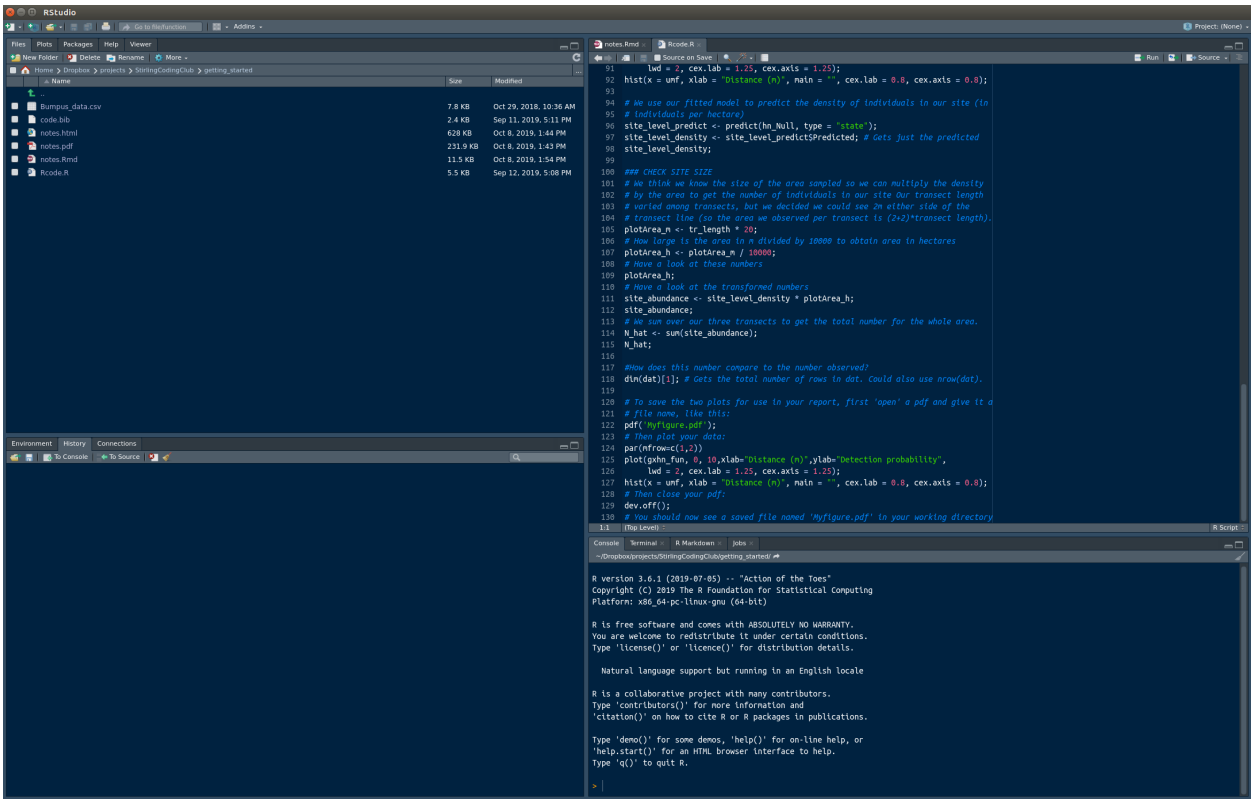
The downside to all of this is that learning R can be a bit daunting at first. Running analyses is not done by pointing and clicking on icons as in Excel, SigmaPlot, or JMP. You need to use code. Here we will start with the very basics and work our way up to some simple data analyses.

Getting started in R, and the basics

Installation. The first thing to do is download Rstudio if you have not already (but see below if you're eager to get started and want to skip this step). Note that R and Rstudio are not the same thing; R is a language for scientific computing, and can be used outside of Rstudio. Rstudio is a very useful tool for coding in the R language. As a very loose analogy, R is like a written language (e.g., English, Spanish) that can be used to write inside Rstudio (e.g., a word processor such as Microsoft Word, LibreOffice). Look carefully at the version of Rstudio that you download; different installers exist for Windows, Mac, and Linux. The most recent version of Rstudio requires a 64-bit operating system. Unless your computer is quite old (say, over seven years), you most likely have a 64-bit operating system rather than a 32-bit operating system, but if you are uncertain, then it is best to check.

Bypassing installation with Rstudio Cloud. If you do not want to install R or Rstudio, or are having trouble doing so but want to get started in R quickly, then an alternative is to use R through the Rstudio cloud (https://rstudio.cloud). The Rstudio cloud allows you to run R right from your browser, and you can sign up for free. You can watch this five minute video to see how to sign up and get started.

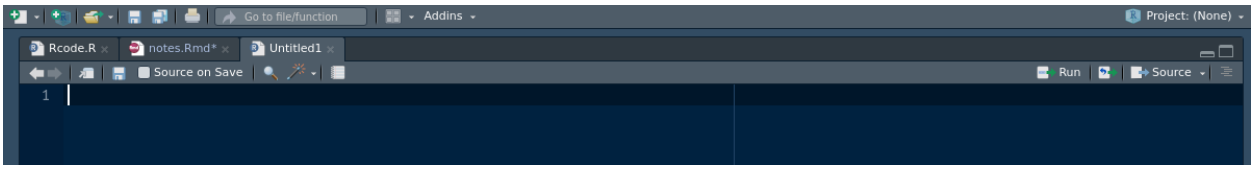
Running Rstudio. When you first run Rstudio, you will see several windows open. It will look something like the below, except probably with a standard black on white theme (if you want, you can change this by selecting from the toolbar 'Tools > Global Options...', then selecting the 'Appearance' tab on the left).



This might look a bit intimidating at first. Unlike Microsoft Excel, SigmaPlot, or JMP, there is no spreadsheet that opens up for you. You interact with R mostly by typing lines of commands rather than using a mouse to point and click on different options. Eventually, this will feel liberating, but at first it will probably feel overwhelming. First, let us look at all of the four panes in the Figure above. Your panes might be organised a bit differently, but the important ones to start out with are the 'Source' and the 'Console'. These are shown in the right hand panes in the above Figure.

To make sure that the Source pane is available to you, open an new R script by selecting from the toolbar 'File > New File > Rscript' (shortcut: Shift+Ctrl+N). You should see a new Rscript open up that looks

something like the below (again, the colour scheme might differ).



Think of this Source file like a Word document that you have just opened up – completely blank and ready for typing new lines of commands to read in data and run analyses. We will come back to this Source file, but for now just know that the Source file stores commands that we want R to intepret and use. The Source file does this by sending commands to the R console, which we will look at now.

The R console should be located somewhere in Rstudio (I like to keep it directly underneath my R Source files). You can identify it by finding the standard R information printed off, which should look something like the below.

```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

The console is where all of the code is run. To get started, you can actually ignore everything else and just focus on this pane. If you click to the right of the greater than sign `>` at the bottom, then you can start using R right in the console. To get a feel for running code in the console, you can use the R console as a standard calculator. Try typing something like the below to the right of the `>`, then hit 'Return' on your keyboard (note, all of my semi-colons are optional – you do not actually need to put them at the end of each line).

```
2 + 5;
```

```
## [1] 7
```

Now try some other common mathematical operations, one line at a time.

```
4 * 4;
```

```
## [1] 16
```

```
12 - 3;
```

```
## [1] 9
```

```
5^2;
```

```
## [1] 25
```

Notice that R does the calculation of each of the above mathematical operations and returns the correct value on the line below. If you are familiar with using Microsoft Excel, this is the equivalent to typing [`= 2 + 5`], or [`= 4 * 4`], etc., into a cell of an Excel spreadsheet. You might also be familiar with spreadsheet functions as well, such as the square root function, which you could use in Excel by typing, e.g., [`= sqrt(25)`] into a spreadsheet cell. This works in the R console too; the functions actually have the same syntax, so you could type the below into the console and hit 'Enter'.

```
sqrt(256);
```

```
## [1] 16
```

The console returns the correct answer 16. Similar functions exist for logarithms (`log`) and trigonmetric functions (e.g., `sin`, `cos`), as they do in Microsoft Excel. But this is just the beginning in R. Functions can be used to do any number of tasks. Some of these functions are built into the base R language; others are written by researchers and distributed in R packages, but you can also learn to write your own R functions to do any number of customised tasks. You will need to use functions in nearly every line of code you write (technically, even the `+`, `-`, etc., are also functions), so it is good to know the basics of how to use them.

Most functions are called with open and closed parentheses, as in the `sqrt(256)` above. The `sqrt` is the function, while the `256` is a function argument. An argument is a specific input to a function, and functions can take any number of arguments. For the `sqrt` function, only one argument is needed, but many arguments will have more than one argument. For example, if we want to take the logarithm of some number using the `log` function, we might need to specify the base. In this case, we clarify the number for which we want to compute the log (`x`) and the logarithm base (`base`). Let us say that we want to compute the logarithm of 256 in base 10.

```
log(x = 256, base = 10);
```

```
## [1] 2.40824
```

Note that some arguments are required, while some are optional. In the case of `log`, the first argument is required, but the base is actually optional. If we do not specify a `base`, then the function simply defaults to calculating the natural logarithm (i.e., base e). Hence, the below also works (note that we get a different answer because the bases differ).

```
log(x = 256);
```

```
## [1] 5.545177
```

In fact, we do not even need to specify the `x` because only one argument is needed for the `log` function. Hence, if only one argument is specified, the function just assumes that this argument is `x`. Try the below.

```
log(256);
```

```
## [1] 5.545177
```

Note that functions can be nested inside other functions, though this can get messy. For example, if you wanted to get the logarithm of the logarithm of 256, then you could write the below.

```
log( log(256) );
```

```
## [1] 1.712929
```

Also note that functions do not need to be mathematical in R; they do not even need to operate on numbers. One very useful function is the `help` function, which provides documentation for other R functions. If, for example, we were not sure what `log` did, or what arguments it accepted, then we could run the code below.

```
help(topic = log);
```

Try running the above line of code in the R console. You should see a description of the `log` function, along with some examples of how it is used and the two arguments (`x` and `base`) that it accepts. Anytime you get stuck with a function, you should be able to use the `help` function for clarification. You can even use a shortcut that returns the same as the `help(topic = log);` above.

```
?log;
```

We now have looked at three functions, `sqrt`, `log`, and `help`. If you have previous experience with Microsoft Excel spreadsheets, you should now be able to make the conceptual connection between typing `=sqrt(25)` into a spreadsheet cell and `sqrt(25)` into the R console. You should also recognise that R functions serve a much broader set of purposes in R. Next, we will move onto assigning variables in the R console.

Assigning variables in the R console

In R, we can also assign values to variables using the characters `<-` to make an arrow. Say, for example, that we wanted to make `var_1` equal 10.

```
var_1 <- 10;
```

We can now use `var_1` in the console.

```
var_1 * 5; # Multiplying the variable by 5
```

```
## [1] 50
```

Note that the correct value of 50 is returned because `var_1` equals 10. Also note the comment left after the `#` key. In R, anything that comes after `#` on a line is a comment that R ignores. Comments are ways of explaining in plain words what the code is doing, or drawing attention to important notes about the code.

Note that we can assign multiple things to a single variable. Here is a vector of numbers created using the `c` function, which combines multiple arguments into a vector or list. Below, we combine six numbers to form a vector called `vector_1`.

```
vector_1 <- c(5, 1, 3, 5, 7, 11); # Six numbers
```

We can now print and perform operations on `vector_1`.

```
vector_1;      # Prints out the vector
```

```
## [1]  5  1  3  5  7 11
```

```
vector_1 + 10; # Prints the vector elements plus ten
```

```
## [1] 15 11 13 15 17 21
```

```
vector_1 * 2;  # Prints the vector elements times two
```

```
## [1] 10  2  6 10 14 22
```

```
vector_1 <- c(vector_1, 31, 100); # Appends the vector
vector_1;
```

```
## [1]   5   1   3   5   7  11  31 100
```

We can also assign lists, matrices, or other types of objects using the `list` function.

```
object_1 <- list(vector_1, 54, "string of words");
object_1;
```

```
## [[1]]
## [1]   5   1   3   5   7  11  31 100
##
## [[2]]
## [1] 54
##
## [[3]]
## [1] "string of words"
```

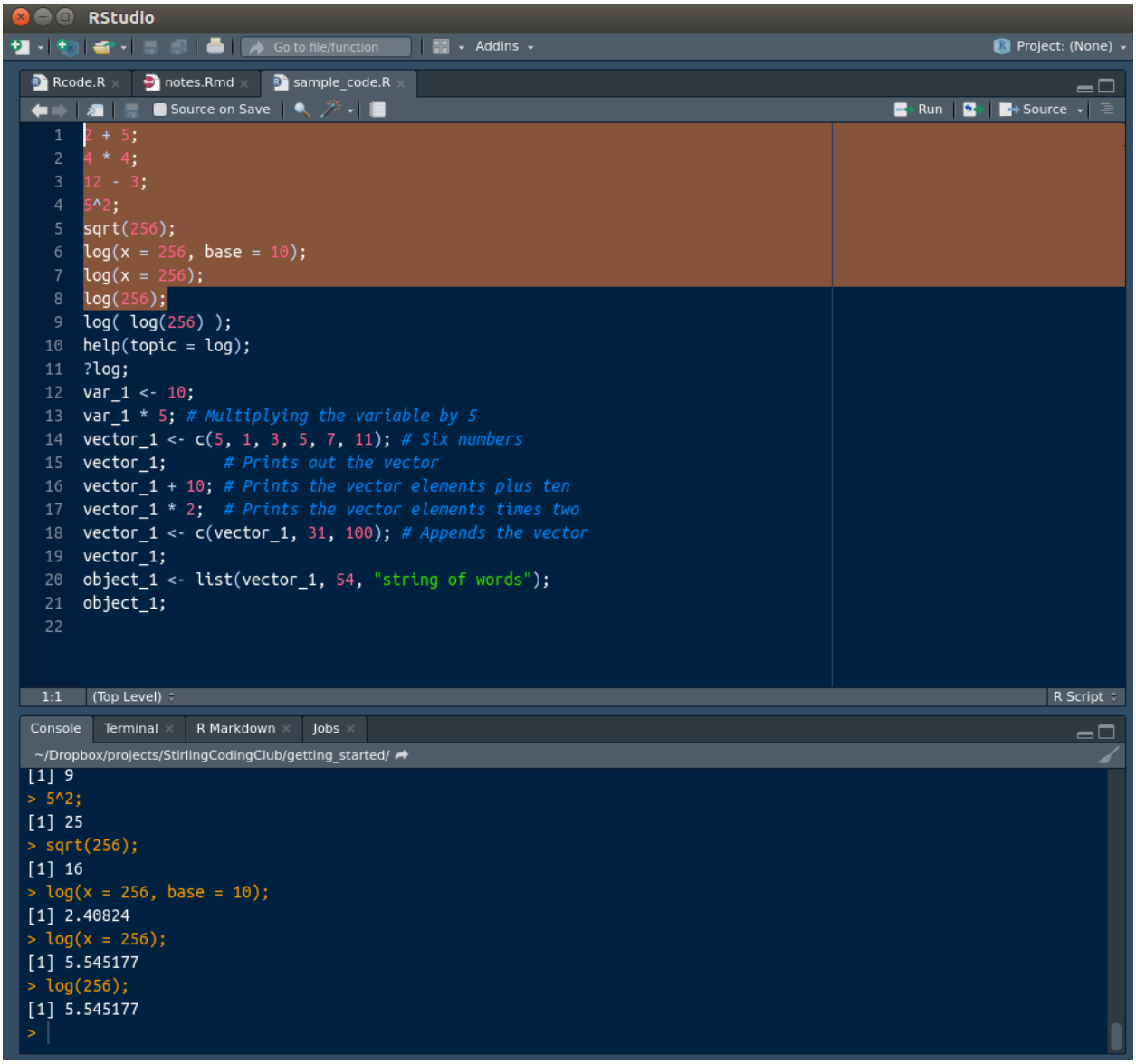
Play around a bit with R before moving on, and try to get confortable using the console. When you have finished with the R console, continue reading to learn how to store lines of code using an R script.

Using R script to save and run code

Up until now, we have focused on running code directly into the console. This works, but if you want to run multiple lines of code, or just save your code for later use, then you will need more than the console. R scripts are plain text files with a '.R' extension, which can be used to save R code. The R code itself is no different than what we have already run into the console. For example, we could save an R file with all of the code that we have read into the console up to this point; it would look like the below.

```
2 + 5;
4 * 4;
12 - 3;
5^2;
sqrt(256);
log(x = 256, base = 10);
log(x = 256);
log(256);
log( log(256) );
help(topic = log);
?log;
var_1 <- 10;
var_1 * 5; # Multiplying the variable by 5
vector_1 <- c(5, 1, 3, 5, 7, 11); # Six numbers
vector_1;      # Prints out the vector
vector_1 + 10; # Prints the vector elements plus ten
vector_1 * 2;  # Prints the vector elements times two
vector_1 <- c(vector_1, 31, 100); # Appends the vector
vector_1;
object_1 <- list(vector_1, 54, "string of words");
object_1;
```

You can find and download the code above in the file sample_code.R on GitHub. If you wanted to redo all of the calculations in this R script, you could open it and run each line one by one, or as a group. In the figure below, I have read sample_code.R into R by first saving it to my computer, then from the toolbar selecting 'File > Open File...' and opening it from the saved location.



There are a few things to note in the Figure above. First, the script sample_code.R now sits above the console; the position of the script might be different depending on your pane settings, but you should be able to see it appear somewhere after opening the file. Second, note how different parts of the text in the R script are coloured differently; this makes reading the text a bit easier. The variables, assignments, arguments, and functions, all appear in white. Numbers are shown in pink, strings of words are in green, and comments are in light blue (your colours might differ, but you should see some distinction among different types of text). Third, note the 'Run' button in the upper right corner above the script. This allows you to run the commands in the R script lines directly into the R console so that you do not have to retype them directly.

You can read the code from the Rscript into the console in multiple ways. The easiest is to simply click with your mouse on whatever line you want to run, then click the 'Run' button. Try clicking anywhere on line 1, for example, so that the cursor is blinking somewhere on the line. Then click 'Run'; you should see `> 2 + 5` appear in the R console, followed by the correct answer 7. After you have done this, R moves the cursor to the next line in anticipation of you wanting to run line 2. If you want to run line 2, then you could just hit 'Run' again, and repeat for line 3, 4, etc. Give this a try.

If you do not want to go through all of the code line by line, you could instead highlight a block of code, as I did above for lines 1-8. If you highlight these lines, then click 'Run', the R console will run every line

one after another, producing the output shown in the console of the Figure above. Try this as well to get a feel for running multiple lines of code at once. You now know the basics of getting started with coding in R. Next, we will move onto reading data into R for analysis, and doing a very simple correlation analysis on the classic Bumpus data set (Bumpus 1898; Johnston, Niles, and Rohwer 1972). Very briefly, the Bumpus data includes characteristics and morphological measurements from sparrows in North America following a severe storm (the specifics are not important for our purposes).

Reading in data

Now we need to read in the Bumpus data. This is actually a challenging part because the data need to be in a correct format and location to be read into R successfully. The format is best read in as a CSV file, though other formats are also possible (TXT can work, or XLSX if you download and load the `openxlsx` R package and use the `read.xlsx` function). For now, I will use a CSV file with the Bumpus data set. Reading CSV files into R can be challenging at first, and I encourage you to first read in the example data set, then try reading in your own data sets into R. You can read your own dat sets into R by saving them as CSV files in Excel; as a general rule, it is good to avoid spaces in these files (replacing them, e.g., with an underscore '_'). Also make sure that all rows and columns are filled in; any empty values can be replaced with an `NA`, which R reads as unavailable data. See the Bumpus CSV file online to get an idea of what a data file looks like.

Two common errors arise at this point, which can be sources of frustration for getting started. First, the data might not be organised correctly for reading into R. Note that rows and columns should start in row '1' and column 'A' of Excel (i.e., don't leave empty rows and columns), and additional cells should not be used outside of rows and columns (if, e.g., you have a value in cell M4, when the last column in your table is K, then R will intepret this as column L being full of empty values). You should be fine if R includes some number of completely filled in rows and columns, with nothing filled in outside. If you want to, you can download the Bumpus CSV file from Dropbox and open it up in Excel for an example of what a good CSV file looks like. Note that there are no empty cells inside the table, and no values outside the table. This should therefore be read easily into R.

Second, you need to make sure that the file you are trying to read into R is located in the same place as your current working directory. You can see what your current working directory (i.e., 'folder') is using the command below.

```
getwd(); # No argument is needed here for the function
```

```
## [1] "/home/brad/Dropbox/projects/StirlingCodingClub/getting_started"
```

The above function returns the current working directory. If this is the same as the CSV file that you want to read into R, then all is well. But if this is not the working directory where your CSV file is located, then you need to find it. You could do this from the R console, but the easiest way is to go to the toolbar and go to 'Session > Set Working Directory > Choose Directory...' and find the location where your CSV file is saved. The easiest way to do this is to save your data in the same place that you have saved your R script. If you do this, then you can simply go to 'Session > Set Working Directory > To Source File Location', and R will set the directory to the same file as your current R script. From there you can read in your CSV file with the `read.csv` function in R. Note that the first row of the file 'Bumpus_data.csv' is a header, which gives the column names, so we should specify the argument 'header = TRUE'.

```
dat <- read.csv(file = "Bumpus_data.csv", header = TRUE);
```

If you get an error message, double-check that the file name and the working directory are correct (if there is an error, this is the problem most of the time). Note that the **everything in R is case sensitive**. That

means that if a letter is capitalised in the file name, but you do not capitalise it in the `file` argument above, then R will not recognise it. A lot of errors are caused by capitalisation issues in R.

Once you have succeeded in reading in a file without getting an error message, to make sure that everything looks correct, you can type `dat` in the console to see all of the data print out. I will use the 'head' function below to just print off the first six rows.

```
head(dat);
```

```
##    sex  surv totlen wingext  wgt head humer femur tibio skull stern
## 1 male alive    154     241 24.5 31.2 0.687 0.668 1.022 0.587 0.830
## 2 male alive    160     252 26.9 30.8 0.736 0.709 1.180 0.602 0.841
## 3 male alive    155     243 26.9 30.6 0.733 0.704 1.151 0.602 0.846
## 4 male alive    154     245 24.3 31.7 0.741 0.688 1.146 0.584 0.839
## 5 male alive    156     247 24.1 31.5 0.715 0.706 1.129 0.575 0.821
## 6 male alive    161     253 26.5 31.8 0.780 0.743 1.144 0.607 0.893
```

If the data appear to be read into R correctly, then you can move on to working with the data and performing analyses in R. Note that `dat` is a big table that is now read into R. While we do not necessarily see the entire table at once, as we would in Excel, we can pull out any of the information in that we want. For example, if we want to see how many rows and columns are in `dat`, we can use the following functions.

```
nrow(dat);
```

```
## [1] 136
```

```
ncol(dat);
```

```
## [1] 11
```

We could also just use the function `dim` to get the dimensions of `dat` (note that this would work for an array of any number of dimensions).

```
dim(dat);
```

```
## [1] 136  11
```

So we know that our table `dat`, which contains the Bumpus data, includes 136 rows and 11 columns. Having read this table into R successfully, we can now perform any number of statistical analyses on the contents. The different ways to analyse theses data are beyond the scope of these notes, but there are a few useful things to know. First, the row and columns in `dat` can be indexed using square brackets. If, for example, we wanted to just look at the value of the fourth row and sixth column, we could type the following.

```
dat[4, 6]; # First row, second column
```

```
## [1] 31.7
```

The first position within the brackets is the row (`4`), and the second position is the column (`6`). Note that R is not restricted to two dimensions; it is possible to have three or more dimensions of an array, in which case we might refer to an array element as `dat[x_dim, y_dim, z_dim]` for a `dat` of three dimensions. Note that we can also store any particular value in `dat` as a variable, if we want. We could, for example store the above as `dat_point_1` using the code below.

```
dat_point_1 <- dat[4, 6];
```

We could then use `dat_point_1` in place of `dat[4, 6]`. We can also define entire rows or columns. For example, if we wanted to return all of the values of row 4, then we could leaven the second index blank, as below.

```
dat[4, ]; # Note the empty space where a column was previously
```

```
##    sex  surv totlen wingext  wgt head humer femur tibio skull stern
## 4 male alive    154     245 24.3 31.7 0.741 0.688 1.146 0.584 0.839
```

In the Bumpus data set, this gives us all the information of measurements for sparrow number 4. We can do the same for columns. Note that column 5 holds the mass of each sparrow (in grams). We could look at all of the sparrow masses using the code below.

```
dat[, 5]; # Note the empty space is now where a row used to be.
```

```
##   [1] 24.5 26.9 26.9 24.3 24.1 26.5 24.6 24.2 23.6 26.2 26.2 24.8 25.4 23.7 25.7
##  [16] 25.7 26.5 26.7 23.9 24.7 28.0 27.9 25.9 25.7 26.6 23.2 25.7 26.3 24.3 26.7
##  [31] 24.9 23.8 25.6 27.0 24.7 26.5 26.1 25.6 25.9 25.5 27.6 25.8 24.9 26.0 26.5
##  [46] 26.0 27.1 25.1 26.0 25.6 25.0 24.6 25.0 26.0 28.3 24.6 27.5 31.0 28.3 24.6
##  [61] 25.5 24.8 26.3 24.4 23.3 26.7 26.4 26.9 24.3 27.0 26.8 24.9 26.1 26.6 23.3
##  [76] 24.2 26.8 23.5 26.9 28.6 24.7 27.3 25.7 29.0 25.0 27.5 26.0 25.3 22.6 25.1
##  [91] 23.2 24.4 25.1 24.6 24.0 24.2 24.9 24.1 24.0 26.0 24.9 25.5 23.4 25.9 24.2
## [106] 24.2 27.4 24.0 26.3 25.8 26.0 23.2 26.5 24.2 26.9 27.7 23.9 26.1 24.6 23.6
## [121] 26.0 25.0 24.8 22.8 24.8 24.6 30.5 24.8 23.9 24.7 26.9 22.6 26.1 24.8 26.2
## [136] 26.1
```

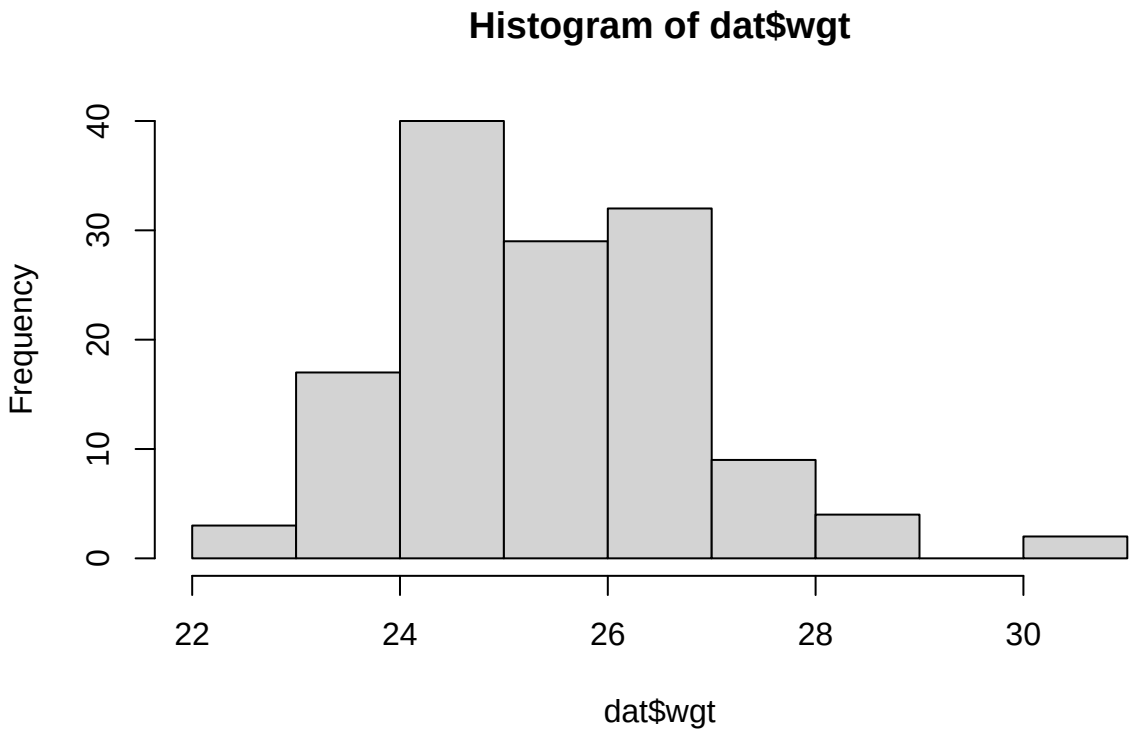
Note that this now returns the masses of all 136 sparrows. Since our table has headers, and the header for column 5 is wgt, we could also use the code below.

```
dat$wgt; # R sees the column header and returns column 5; same as above
```

```
##   [1] 24.5 26.9 26.9 24.3 24.1 26.5 24.6 24.2 23.6 26.2 26.2 24.8 25.4 23.7 25.7
##  [16] 25.7 26.5 26.7 23.9 24.7 28.0 27.9 25.9 25.7 26.6 23.2 25.7 26.3 24.3 26.7
##  [31] 24.9 23.8 25.6 27.0 24.7 26.5 26.1 25.6 25.9 25.5 27.6 25.8 24.9 26.0 26.5
##  [46] 26.0 27.1 25.1 26.0 25.6 25.0 24.6 25.0 26.0 28.3 24.6 27.5 31.0 28.3 24.6
##  [61] 25.5 24.8 26.3 24.4 23.3 26.7 26.4 26.9 24.3 27.0 26.8 24.9 26.1 26.6 23.3
##  [76] 24.2 26.8 23.5 26.9 28.6 24.7 27.3 25.7 29.0 25.0 27.5 26.0 25.3 22.6 25.1
##  [91] 23.2 24.4 25.1 24.6 24.0 24.2 24.9 24.1 24.0 26.0 24.9 25.5 23.4 25.9 24.2
## [106] 24.2 27.4 24.0 26.3 25.8 26.0 23.2 26.5 24.2 26.9 27.7 23.9 26.1 24.6 23.6
## [121] 26.0 25.0 24.8 22.8 24.8 24.6 30.5 24.8 23.9 24.7 26.9 22.6 26.1 24.8 26.2
## [136] 26.1
```

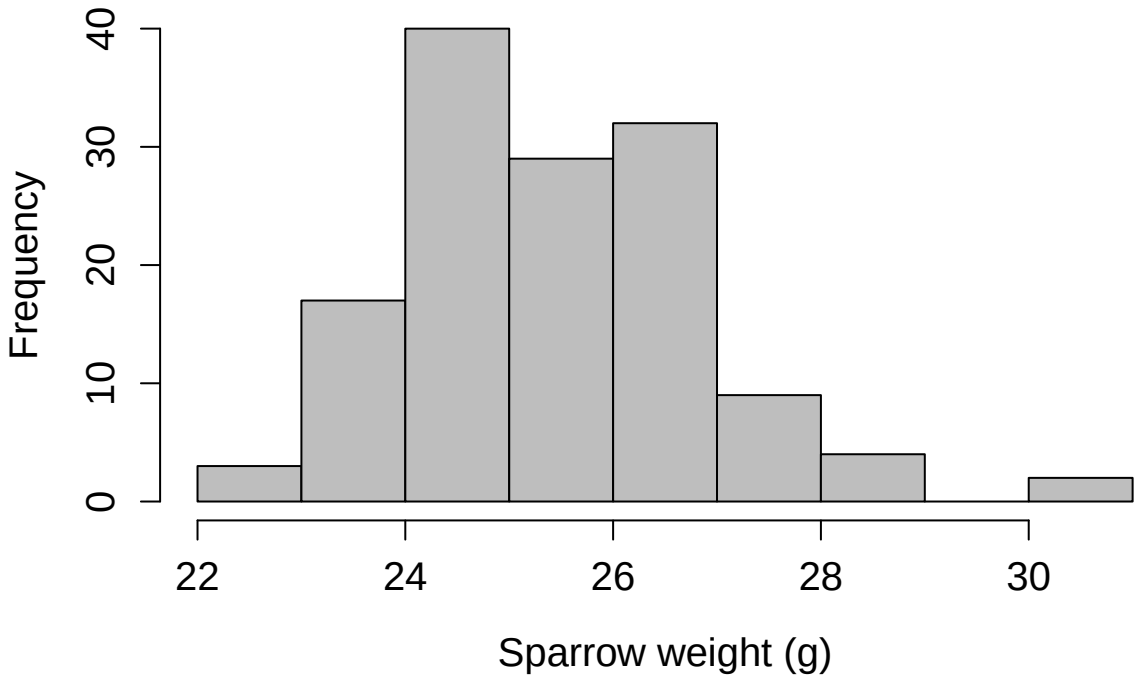
Even better, we can plot a histogram of sparrow weights using the built-in function `hist` in R.

```
hist(x = dat$wgt);
```



This looks a bit rubbish; the main title is unnecessary, and the axis labels are not terribly informative. We can tweak the axis labels and colours using the following arguments.

```
hist(x = dat$wgt, main = "", xlab = "Sparrow weight (g)", ylab = "Frequency",
     cex.lab = 1.25, cex.axis = 1.25, col = "grey");
```



Some basic analyses

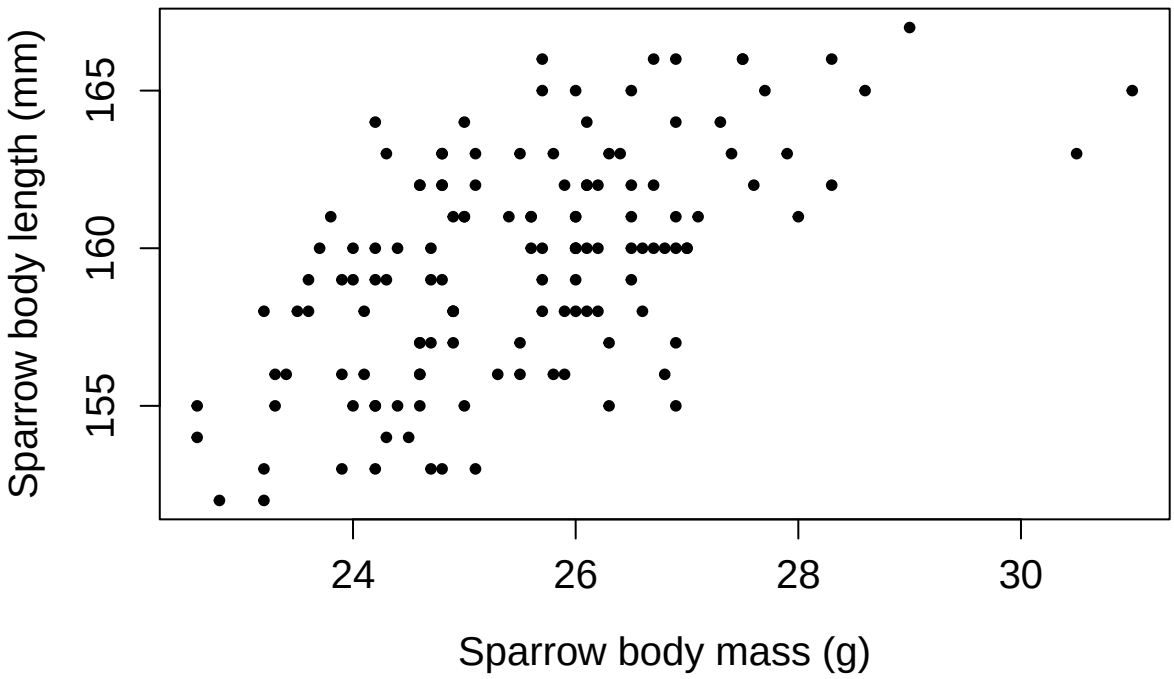
Since the R programming language is developed primarily for scientific analysis and statistical computing, there are several built-in functions for doing simple analyses. More complex analyses that are not possible with built-in functions can be performed by downloading R packages (this can be done from the Comprehensive R Archive Network (CRAN) using the funciton `install.packages`, or by looking at the packages tab in Rstudio). If you have heard of an analysis, then it is probably available within an R package. For now, let us just do some basic statistical analyses. For example, let us say that we want to summarise the data on sparrow body mass. We can do this with the `summary` function in R.

```
summary(dat$wgt);
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   22.60   24.57   25.55   25.52   26.50   31.00
```

The `summary` function returns six numbers, including the minimum value, first quartile, median, mean, third quartile, and maximum value of numbers in the data, in this case a column of sparrow body masses. Now say that we want to see if sparrow body mass is correlated with sparrow body length (`dat$totlen` in our data frame). We could first look at a scatter plot of body mass versus length.

```
plot(x = dat$wgt, y = dat$totlen, xlab = "Sparrow body mass (g)",
     ylab = "Sparrow body length (mm)", cex.lab = 1.25, cex.axis = 1.25,
     pch = 20); # Note: cex.lab, cex.axis, and pch are purely cosmetic
```



It clearly looks like there is a correlation between the two variables of interest. We can find out what this correlation is below.

```
cor(dat$wgt, dat$totlen);
```

```
## [1] 0.5838648
```

Hence, the correlation between sparrow body mass and sparrow total body length is 0.5838648. We can even test to see if this correlation is significant using the `cor.test` function.

```
cor.test(dat$wgt, dat$totlen);
```

```
##
##  Pearson's product-moment correlation
##
## data:  dat$wgt and dat$totlen
## t = 8.3251, df = 134, p-value = 8.612e-14
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.4608233 0.6848847
## sample estimates:
##       cor
## 0.5838648
```

As you can see above, this returns the correlation, along with a t-statistic, degrees of freedom, p-value, and confidence interval for the Pearson product-moment correlation between the two variables of interest. If we wanted to instead run a simple linear regression of body total length against body mass, we could use the `lm` function as below.

```
lm(dat$totlen ~ dat$wgt);
```

```
##
## Call:
## lm(formula = dat$totlen ~ dat$wgt)
##
## Coefficients:
## (Intercept)      dat$wgt
##     123.571        1.409
```

Note that the above function returns the intercept and slope, but not any results from statistical null hypothesis tests. To do this, we need to wrap `lm` in the function `summary`, as below (the R function `summary` can tell the difference between a vector of numbers and a model, and handles each differently).

```
summary( lm(dat$totlen ~ dat$wgt) );
```

```
##
## Call:
## lm(formula = dat$totlen ~ dat$wgt)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.4819 -2.1078 -0.2135  2.1958  6.3232
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 123.5713     4.3282  28.550  < 2e-16 ***
## dat$wgt       1.4093     0.1693   8.325 8.61e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 2.902 on 134 degrees of freedom
## Multiple R-squared:  0.3409, Adjusted R-squared:  0.336
## F-statistic: 69.31 on 1 and 134 DF,  p-value: 8.612e-14
```

Now we get a bit more information, including significance tests for our intercept `(Intercept)` and slope `dat$wgt`. The linear model function (`lm`), and the generalised linear model function (`glm`) are very flexible, and can be used for a variety of purposes that I will not elaborate on here, except for one more example. Let us look at the categorical variable of sparrow sex, and test whether or not bird mass difference between sexes. To do this, we could use a simple t-test with the `t.test` function below. For illustrative purposes, I have set the argument `var.equal` equal to `TRUE` to assume equal variances (had I not done this, the default would have been `var.equal = FALSE`).

```
t.test(dat$wgt ~ dat$sex, var.equal = TRUE);
```

```
##
##  Two Sample t-test
##
## data:  dat$wgt by dat$sex
## t = -3.0333, df = 134, p-value = 0.002906
## alternative hypothesis: true difference in means between group female and group male is not equal to
## 95 percent confidence interval:
##  -1.2820247 -0.2700279
## sample estimates:
## mean in group female   mean in group male
##             25.02857             25.80460
```

Since a t-test is equivalent to a linear model with two categorical response variables, we could also use the `lm` function to do the same null hypothesis test.

```
summary( lm(dat$wgt ~ dat$sex) );
```

```
##
## Call:
## lm(formula = dat$wgt ~ dat$sex)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.6046 -1.0286 -0.1046  0.9144  5.4714
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  25.0286     0.2046 122.316  < 2e-16 ***
## dat$sexmale   0.7760     0.2558   3.033  0.00291 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.432 on 134 degrees of freedom
## Multiple R-squared:  0.06425,    Adjusted R-squared:  0.05727
## F-statistic: 9.201 on 1 and 134 DF,  p-value: 0.002906
```

Or we could use `aov` to run an (again, equivalent) analysis of variance (ANOVA).

```
summary( aov(dat$wgt ~ dat$sex) );
```

```
##              Df Sum Sq Mean Sq F value  Pr(>F)
## dat$sex       1  18.88  18.877   9.201 0.00291 **
## Residuals   134 274.92   2.052
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note that the p-value is identical in each of the above three cases because the underlying mathematics is the same. The point here is not to demonstrate all of the different statistics that can be performed within R. Because R is a programming language, the possibilities are limitless; in the unlikely chance that no one has written code for a particular statistical analysis that you need to perform, you could write your own code to do the analysis yourself.

Practicing with the Bumpus data set, or your own

If you want to continue practicing working in R, then you can download the Bumpus data set here (right click on 'Bumpus_data.csv' and select 'Save link as...') or here (select 'Open with' in the upper right to download). Save the file somewhere on your computer, then make sure that you set R to the same working directory as the saved file by going to the toolbar of Rstudio and selecting 'Session > Set Working Directory > To Source File Location' (note, you could also do this from the R console using the `setwd` function). You should now be able to read these data into R and start doing some analyses. I have recreated the above analysis with the Bumpus data set into an Rscript. You can download it here, or recreate it yourself by copying and pasting the code from the appendix below into a new Rscript.

Appendix: R script file

```
# First we need to read in the Bumpus data below
# Make sure that R is set to the same working directory as Bumpus_data.csv
# Also make sure that the filenames match exact (including capitalisation)
dat <- read.csv(file = "Bumpus_data.csv", header = TRUE);
# Now the whole data table is saved as the variable 'dat'
# You can save it as something different if you want, but avoid spaces

# Let's take a look at the first six rows of the data
head(x = dat); # Note that the 'x = ' is not necessary. 'head(dat)' works fine

# How many rows and columns are in the data?
dim(x = dat);

dat[4, 6]; # First row, second column?

# The below produces a histogram of sparrow mass
# Note that I break the line below mid-function after specifying 'ylab'. This
# isn't required, but it often makes code more readable to break to a new line
# when the line exceeds 80 characters.
hist(x = dat$wgt, main = "", xlab = "Sparrow weight (g)", ylab = "Frequency",
     cex.lab = 1.25, cex.axis = 1.25, col = "grey");
```

```r
# Let's get a summary of just the sparrow mass
summary(object = dat$wgt);

# Now let's plot sparrow totel length against sparrow mass
plot(x = dat$wgt, y = dat$totlen, xlab = "Sparrow body mass (g)",
     ylab = "Sparrow body length (mm)", cex.lab = 1.25, cex.axis = 1.25,
     pch = 20); # Note: cex.lab, cex.axis, and pch are purely cosmetic

# Test the correlatoin between total length and mass
cor.test(x = dat$wgt, y = dat$totlen);

# Make a linear model of total length regressed against body mass
our_model <- lm(formula = dat$totlen ~ dat$wgt);

# Now summarise 'our_model' that we saved above
summary(object = our_model);

# Now test if bird mass differs by sex using a t-test
t.test(formula = dat$wgt ~ dat$sex, var.equal = TRUE);

# We could also do a linear model get the same results
our_lm <- lm(formula = dat$wgt ~ dat$sex)
summary(object = our_lm);

# And we can do the same with an ANOVA
our_aov <- aov(dat$wgt ~ dat$sex)
summary(our_aov);
```

References

Bumpus, Hermon C. 1898. "Eleventh lecture. The elimination of the unfit as illustrated by the introduced sparrow, *Passer domesticus*. (A fourth contribution to the study of variation.)." *Biological Lectures: Woods Hole Marine Biological Laboratory*, 209–25.

Johnston, R F, D M Niles, and S A Rohwer. 1972. "Hermon Bumpus and natural selection in the House Sparrow *Passer domesticus* ." *Evolution* 26: 20–31.